Scripting with Python

Schrödinger Suite 2006

Schrödinger Press

Revision A, April 2006

Contents

Document Conventions

In addition to the use of italics for names of documents, the font conventions that are used in this document are summarized in the table below.

Font	Example	Use
Sans serif	Project Table	Names of GUI features, such as panels, menus, menu items, buttons, and labels
Monospace	`$SCHRODINGER/maestro`	File names, directory names, commands, environment variables, and screen output
Italic	*filename*	Text that the user must replace with a value
Sans serif uppercase	CTRL+H	Keyboard keys

In descriptions of command syntax, the following UNIX conventions are used: braces `{ }` enclose a choice of required items, square brackets `[ ]` enclose optional items, and the bar symbol `|` separates items in a list from which one item must be chosen. Lines of command syntax that wrap should be interpreted as a single command.

In this document, to *type* text means to type the required text in the specified location, and to *enter* text means to type the required text, then press the ENTER key.

References to literature sources are given in square brackets, like this: [10].

Introduction

Since the beginning, Maestro has had a command language. However, until now, scripting abilities have been limited to simple lists of Maestro commands. Feedback from users has shown us that many people need abilities well beyond that. For this reason Maestro 7.0 provides improved scripting ability with the well-known scripting language Python. The combination of Python, the Maestro command language, and Python functions for using Maestro functionality, is very powerful and greatly expands the user's ability to automate and customize Maestro.

On top of the Maestro interface, the Python tools provided with Schrödinger's software include interfaces to the Job Control facility. With these interfaces, scripting capabilities can be extended into automated workflows that combine Schrödinger's products in ways that suit the user's needs.

Python is easy to use and fun to learn. It is a straightforward language that doesn't require the kind of learning curve normally associated with a programming language.

This document aims to provide a non-technical introduction to using Python within Maestro. This tutorial provides lots of examples and includes a brief description of the most important things you need to know about Python. While a detailed description of how to program in Python is beyond the scope of this tutorial, that information is readily available elsewhere.

About Python

2.1 What is Python?

Python is a feature-rich scripting language that has gained broad acceptance in a wide range of applications. Unlike programming languages such as C, C++ or Fortran (the languages used to develop other Schrödinger software like Maestro itself) Python is what is known as an *interpreted* language. This means that it doesn't have to undergo an extensive compilation process before it can be used. Developing scripts in Python is fast and easy. All the examples given in this document are fairly short and deliberately avoid advanced language features. However, all Python language features are available from within Maestro including the use of nearly all of the hundreds of third party modules[1] available for Python.

What makes Python so valuable for us is that it is both *embeddable* and *extensible*. We have embedded Python in Maestro so that it can provide scripting facilities to control the program. At the same time we have also extended Python by providing access to a whole range of Maestro functionality for dealing with chemical structures, Maestro files, and projects.

2.2 Where Can I Find Out More About Python?

There are a number of excellent sources of information about Python, both online and print materials. Here are a few:

- www.python.org[1] contains documentation, examples and a great many interesting links. Nearly everything you read about Python on this and related web sites will be relevant to your use of Python in Maestro. Version 7.5 of Maestro uses Python 2.3.3 so many recent Python language features are available from within Maestro.
- If you have some experience with scripting or programming but not with Python, Dive into Python[1] is filled with valuable, graduated examples that will help you master Python.
- If you are a more experienced user, Thinking in Python[1] covers more advanced concepts like applying design patterns to Python.
- O'Reilly[1] has published a number of popular Python books.
- The Vaults of Parnassus[1] contain many useful Python modules and examples.

1. Please see the notice regarding third party programs and third party Web sites on the copyright page at the front of this manual.

2.3 Some Useful Things to Know About the Python Language

This section is targeted at those who do not have the time to fully master Python but still want to write (or modify existing) scripts. It focuses on the basic core of Python, just the bare essentials with which to get started.

Indentation matters in Python. Unlike other scripting and programming languages which use { } to group statements together, statements in Python are grouped simply by the level to which they are indented relative to one another. This means you need to take care in order to ensure the indentation reflects the logic you intend. Consider the following two examples:

```
# Example 1:
x=5
y=4
if x > 5 :
    x = x-1
x=y+1
print x
```

This example prints `5` because the line `x=y+1` is not part of the `if` block.

```
# Example 2:
x=5
y=4
if x > 5 :
    x = x-1
    x = y+1
print x
```

This second example prints `5`, since the lines `x=x-1` and `x=y+1` are executed if and only if `x` is indeed greater than `5`. All Python scripts use indentation to indicate grouping of statements, and it is usually considered good form to place each Python statement on its own line.

Note: You can use either space characters or tabs for indents, but you should not mix them. We recommend using spaces.

Number sign (`#`) is used for comments. Comments begin with a `#` and extend to the end of the current line. The Python interpreter ignores comments.

The keyword `def` is used to define functions. Function parameters may have default values. Functions (groups of Python statements called to perform a particular task) are defined using the `def` statement:

```
# Example 3
def myfunc( x, y=10 ):
    x = x + y
   print x


myfunc( 5 )
myfunc( 5, 5 )
```

Here we have defined a function called `myfunc()`. It takes two arguments, `x` and `y` where `y` has a default value of `10`. This means you can call `myfunc` in two ways as shown in this example. In the first call to `myfunc()` a value for `x` is supplied, but no value for `y`. This means the default value for `y` is used. As a result `myfunc` computes the sum of `5` and `10` and prints the result which is `15`. In the second use, values are supplied for both `x` and `y`. By providing an explicit value for `y`, the default value is overridden and the result is the sum of `5` and `5` which is `10`.

Variables do not need to be explicitly typed or destroyed. In the above examples, the types (integer, string, real number) of the variables `x` and `y` are defined only by the context and do not need to be declared before the variable is used. Also, unlike other languages such as C and C++, there is no need to allocate or free memory in Python—this is handled automatically.

All Python script files should have `.py` suffix. Without this suffix, Maestro does not recognize the files as Python scripts.

2.4 Why Python?

Python is not the only embeddable and extensible scripting language available. Ruby, Tcl and Perl are all other possibilities. Perl is probably the most serious other contender. In many cases a preference for a given language is a personal matter, that depends on the user's experience and personal style. However, given that it is technically possible to use Perl in the same way we use Python, and that in terms of language features essentially the same sorts of things are possible with either, there were three reasons we chose Python instead of Perl:

1. While Perl is used widely for system administration and web programming, Python is more popular in scientific programming and larger scale development. Python is very scalable: it can be used to write simple scripts or to develop full-fledged applications.
2. Of the Maestro users surveyed, most expressed either a strong preference for Python over Perl, or no preference at all.
3. Python has a clean, straightforward syntax; a benefit for new or occasional users. Consider the following two fragments, which perform the same task:

Example 1. Perl sample

```
use mmlibs;
use mmlibsext;
use mae;
sub spin {

    # The three input parameters are assigned passed values or the default.
    my $axis = shift || 'Y';
    my $step = shift || 10;
    my $slp = shift || 0.1;
    my $total_rotate=0;

    # If the axis is not X,Y or Z then we can die with an error string
    # which will be posted as a Maestro error dialog.
    if( $axis !~ m/^X|^Y|^Z/i )  {
        die "$axis is not a valid axis value - must use X,Y or Z";
    }

    # Now we begin the code which actually does the rotation. Start with
    # a loop that finishes when we've rotated 360 degrees:

    while( $total_rotate < 360.0 ) {
    # Issue a rotate command. Note that the variables $axis and $step
    # will be replaced with their values before Maestro gets to see
    # the command.
    mae_command("rotate $axis=$step");
    # Redraw the window
    mae_main_window_redraw();
    # Increment the total rotation by the step increment:
    $total_rotate += $step;
    # The following line is not obvious but it appears to be the best
    # (only?) way to pause for a given time from a Perl script.
    select undef, undef, undef, $slp
    }
    return;
```

Example 2. Python sample

```
from schrodinger import maestro
from schrodinger import mm
import time
def spin(axis="Y",step=10,slp=0.1):
    total_rotate=0

    # Verify the axis:
    if axis != "X" and axis != "Y" and axis != "Z" :
        raise "MyException", "Can't use axis: " + axis

    # Now we begin the code which actually does the rotation. Start with
    # a loop that finishes when we've rotated 360 degrees:

    while( total_rotate < 360.0 ) :
        # Issue a rotate command. Note that the %s and %d
        # will be replaced with their values before Maestro gets to see
        # the command.
        maestro.command("rotate %s=%d" % (axis,step))
        # Redraw the window
        maestro.redraw()
        # Increment the total rotation by the step increment:
        total_rotate += step
        time.sleep(slp)
    return
```

In language comparisons like this, there really is no "right" answer. However, while you may find some aspects of Python a little odd at first, we urge you to persist. Python is easy to learn and easy to use.

There are a number of accounts available of programmers who have made the switch from Perl to Python. One of the best is the Why Python?[2] essay, by well-known programmer and author Eric S. Raymond.

2.5 Isn't It Too Slow?

One of the limitations of an interpreted language like Python is that because each line needs to be parsed and “understood” (interpreted) at the time the script is run, it tends to be slow in comparison to compiled languages like C, which are translated into a low-level machine description.

2. Please see the notice regarding third party programs and third party Web sites on the copyright page at the front of this manual.

However, one important point about the way we use Python, both in Maestro and outside of it, is that much of the heavyweight computation and manipulation of chemical structures is not actually done in Python. As you will see from the examples in this document, Python in Maestro is primarily used for controlling the program. The real work is actually done by Maestro itself. The time and overhead required to interpret a Python script is generally insignificant relative to the work done by Maestro to execute the commands sent to it from a Python script.

Running Python Within Maestro

3.1 Overview - What Can I Do With Python in Maestro?

There are two main ways that Python scripts can interact with Maestro: issue Maestro commands and manipulate Workspace structures and Project Table entries. It is important to realize that, although Python is the *scripting* language for Maestro in v7.0, there is no change to the *command* language. The syntax of Maestro commands remains the same. However, what *has* changed is the ability to issue Maestro commands from Python, a feature that greatly increases the utility of these commands. Python scripts can contain control structures (if, while, for, etc.), variables and parameters (values supplied by the user when the script is run), so now it is possible to write scripts that are considerably more flexible and general than the simple lists of Maestro commands previously available.

While issuing commands with Python scripts is an extremely powerful mechanism for controlling Maestro, it is essentially a one-way communication. There is no way for the scripts to receive information about Maestro, the structure in the Workspace or entries in the Project Table. Therefore we have introduced an additional mechanism so Python scripts can directly manipulate structures and entries, right down to the level of changing the properties of individual atoms. In practice many Python scripts use a combination of these two approaches.

3.2 A First Python Script in Maestro

This tutorial starts with a simple script that illustrates the basic mechanism for creating and running scripts within Maestro.

You can create your Python scripts with any text editor. Some editors, like Emacs, include special capabilities for Python such as colorizing keywords, comments, and strings; and maintaining the correct indentation. Another possibility is the standard Python editor *idle*, which is available at `$SCHRODINGER/utilities/idle`.

Example 1. myfirst.py

In your editor, create the following script and name the file `myfirst.py`:

```
# myfirst.py
def myfirst():
    print "Hello World from Maestro"
```

Take care to indent the second line. It indicates that `print "Hello World from Maestro"` is part of the `myfirst()` function. Save your script and start Maestro from the directory in which you saved your script.

3.3 Scripts, Modules, and Functions

Before we actually run the script in Maestro, it is worth discussing the terminology surrounding Python scripts. While we will continue to use the term *script* in fairly loose terms, technically what you just created is a Python *module*. Modules are named after the files that contain them. Since you named your file `myfirst.py`, the module is called `myfirst`.

Modules generally contain a number of functions. In this example, our module contains a single function, also named `myfirst()`. There is, however, no reason why a function needs to be named after the module: it can have any name you wish.

It is important to understand the distinction between functions and modules. When you ask Maestro to run a script it is actually executing a single function inside a module; it does not, in general, run the entire contents of the module. While not recommended, you can in fact place all the functions you write into a single file.

A common practice is to place a set of related functions into a module. Some may not be designed to be directly called from Maestro. You might, for example, call one function from Maestro and that function in turn might call other functions in the same module.

We have provided a number of useful functions that you can use as starting points to create your own. You will see more on this in the next chapter.

3.4 The pythonrun Command

Now it is time to run your script. You saved your script as `myfirst.py` and started Maestro from the directory that contains the script. Now, enter the following from the Maestro command line:

```
pythonrun myfirst.myfirst
```

You should see:

```
Hello World from Maestro
```

displayed in the terminal window from which you started Maestro. Congratulations! You just wrote and executed your first Python script from Maestro.

Note: We specified both the module and function as part of the `pythonrun` command.

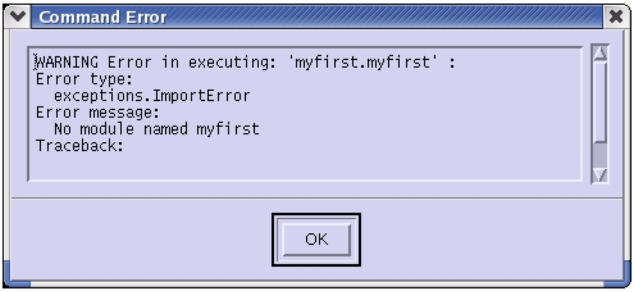

Figure 3.1. Import Error.

In this example, we had direct access to our script by starting Maestro from the directory containing the script, but this can quickly become restrictive. See Section 3.8 on page 13 for details on where to store modules.

3.5 What To Do If It Doesn't Work

If the script did not run as you expected, there are a few things you can try. The error message in Figure 3.1 means that Maestro was not able to locate the `myfirst` module. Check the following:

- The file is saved and named `myfirst.py`
- The file exists in the directory from which Maestro was started
- You correctly typed the command as `pythonrun myfirst.myfirst`

If you see an error message like Figure 3.2 you probably introduced a syntax error into the script. For example, this message indicates the indentation was not correct. The second line of the function must be indented relative to the first so that Python knows `print` belongs to the `myfirst` function.

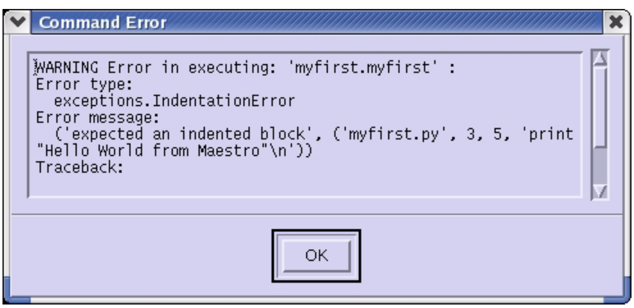

Figure 3.2. Indentation Error.

3.6 The pythonimport Command

While `pythonrun` is the most commonly used Maestro command associated with Python scripting, if you are making frequent changes to a python script you will also find `pythonimport` useful. One of the side effects of using `pythonrun` is that a copy of the module containing the requested script is actually loaded into memory. This allows Maestro to be more efficient in handling repeated requests for the same script. But if you now make changes to the `myfirst.py` file while Maestro is running and once again enter:

```
pythonrun myfirst.myfirst
```

you do not see the effects of your changes. Maestro is still using the original `myfirst` script from the previously loaded `myfirst.py` file. This is where the `pythonimport` command comes into play. When you run:

```
pythonimport myfirst
```

it causes Maestro to reload the `myfirst.py` file so that the next time you execute the script using `pythonrun`, the updated version is used.

3.7 Adding a Parameter

A small change to `myfirst.py` illustrates how to pass information to a Python script in Maestro.

Example 2. myfirst.py

```
# myfirst.py
def myfirst( param = "" ):
   print "Hello World from Maestro " + param
```

Save your changes and then use `pythonimport` to reload `myfirst.py`. Now run the script:

```
pythonrun myfirst.myfirst
```

It displays the same output as the first version of `myfirst.py`: “Hello World from Maestro”. This is because we added an empty string as the default parameter. If you do not supply a value for `param`, the script supplies an empty string. If we had not added the default value `(param = "")`, then running `myfirst` without a value for the parameter results in an error.

Now run the script with a parameter:

```
pythonrun myfirst.myfirst "a second time"
```

You should see:

```
Hello World from Maestro a second time
```

This is a very significant feature of running Python from within Maestro. Unlike older Maestro command scripts, where everything needed to be hard-coded into the script, Python functions can take any number of parameters, some or all of which can have default values. This allows Python scripts to be very general. For example, you can write a script that operates on a file-name supplied by the user at the time the script is run.

Note: With the exception of quoted strings (""), which are treated as a single parameter, all parameters entered from the Maestro command line input area must be separated by white space.

3.8 Module Search Path

This section describes how Maestro locates the module file. In our example we made sure that Maestro was started from the directory containing `myfirst.py`. This is typical for developing scripts. However, it is not very convenient when you want to use your script in many different directories. To make this easier, we created a special directory for your scripts:

your home dir`/.schrodinger/maestro`*version*`/scripts`

where *version* is the 2-digit Maestro version number. Inside this directory you can place files like `myfirst.py`. Then you can issue commands like:

```
pythonrun myfirst.myfirst
```

from any directory in which you run Maestro.

Note: Maestro looks first in the current directory (the one from which you started Maestro) first before looking in `$HOME/.schrodinger/maestro`*version*`/scripts`. So if you have a local copy of the script, the local copy is used instead.

3.9 The pythoneval Command

There is a third Python-related Maestro command, `pythoneval`. This command is used less frequently than the commands already discussed, but it is described here for completeness.

`Pythoneval` allows you to execute any Python expression in the Maestro Python interpreter. This means you have interactive access to the interpreter.

You can achieve the same result as `pythonrun` but you must remember to first import the module and use the Python style syntax to call any functions. So using the example for `myfirst.py` as described above:

```
pythoneval import myfirst
pythoneval myfirst.myfirst("a second time")
```

is equivalent to:

```
pythonrun myfirst.myfirst "a second time"
```

Now you know the basics of creating Python scripts and running them from within Maestro. The next chapter describes how to create a Python script that actually interacts with Maestro.

Issuing Maestro Commands

The previous chapter showed how to run a very simple Python script from within Maestro and introduced modules and functions. This chapter discusses how you can write scripts to issue Maestro commands.

4.1 The maestro Python Module

In order to communicate with Maestro you need to first import the provided `maestro` module into your script. You'll see that most of the example scripts in the remainder of this document contain a line which looks something like:

```
from schrodinger import maestro
```

near the top. This tells Python to load and make available the functions contained in the `maestro` module.

There are several variants of this. In our examples we use the `from schrodinger import maestro` form. This means that in the example scripts we will always reference the functions in this module in the fully qualified form as `maestro.`*function*`()`. However, if you look at Python documentation or other scripts, you will see that there are other ways you can use the `import` command. For example:

```
from schrodinger.maestro import *
```

This means that everything in the `maestro` module can be used directly in your script without the `maestro.` prefix.

You can even go further and use an import expression such as:

```
from schrodinger.maestro import command as c
```

This means that instead of using `maestro.command()` you can simply use `c()`.

Note: You can write your own modules and import them into other modules. The only requirement is that your modules are in the module search path as described in the previous chapter.

4.2 Sending a Command to Maestro From a Python Script

Example 1. roty.py

1. Create a file called `roty.py` that contains the following:

```
# roty.py
from schrodinger import maestro

def roty( by=90 ):
    maestro.command( "rotate y=%d" % by )
```

2. In Maestro, place a structure in the Workspace.
3. In the command input area, enter the following command:

   ```
   pythonrun roty.roty
   ```

   You should see the contents of the Workspace rotate by (the default value of) 90 degrees about the Y axis.
4. You can rotate by any amount by supplying your own value for the parameter. For instance:

   ```
   pythonrun roty.roty 30
   ```

   rotates the contents of the Workspace by 30 degrees.

There are a couple of notable things about this script:

- We use `maestro.command()` to tell Maestro to issue a command. The command is just a normal Maestro command (and can include any aliases you may have defined for the command). The hundreds of available commands and their options are documented in the *Maestro Command Reference Manual*.

  Note: You can review the commands that have been issued in the normal operation of Maestro by choosing Command Script Editor from the Edit menu.

- The `maestro.command()` function takes a single string parameter. However we can substitute the value of variables into that string before issuing the command. This is what the

  ```
  "rotate y=%d" % by
  ```

  expression does. The value of `by` is substituted into the string before the command is issued. This is an extremely powerful mechanism and we provide more examples later in this document.

There are a couple of enhancements we can make to this script. In the previous example the axis of rotation is hard coded into the Python function. We could create similar functions called `rotx()` and `rotz()` to rotate around the other axes. Another option is to supply the axis of rotation as a parameter and add that into the command string before issuing the command. Here is the reworked example in which we have not only parameterized the axis of rotation but also taken care to verify that the supplied argument is a valid axis:

```
# rot.py
from schrodinger import maestro

def rot( axis="y", by=90 ):

    if( axis != "y" and axis != "x" and axis != "z" ):
        raise Exception, "%s is not a valid axis" % axis

    maestro.command( "rotate %s=%d" % (axis,by) )
```

Now you can issue a Maestro command such as:

```
pythonrun rot.rot x 30
```

to get a rotation of 30 degrees around the X-axis.

Note: While this might be a good example of building your own function that uses parameters, you will probably want to use the built-in `rotate` command to rotate structures in the Workspace:

```
rotate x=30
```

4.3 Other Ways to Use maestro.command()

There are two other ways in which `maestro.command()` can be used. First, instead of providing a single string, it is possible to specify the keyword, operands, and options of the Maestro command separately. For example, to rotate around the x, y, and z axis use the following:

```
maestro.command("rotate", x=5)
maestro.command("rotate", y=5)
maestro.command("rotate", z=5)
```

In this way the keyword `rotate` is a separate parameter to the options: `x=5`, etc.

It is also possible to issue more than one command in a single call to `maestro.command()` by using a triple quoted string with each command on a separate line. For example, the commands above can also be issued as:

```
maestro.command("""
rotate x=5
rotate y=5
rotate z=5
""")
```

This is a useful way to issue a series of Maestro commands taken directly from the Maestro command script editor. Note that each command *must* be on a new line.

Example 2. spin.py

Before we finish this introduction we will look at a more sophisticated example—a Python function that does something not currently possible with a Maestro command. The following example uses the form of `maestro.command()` where the options are specified separately from the keyword:

```
#spin.py
from schrodinger import maestro
import time
def spin(axis="Y",step=10,slp=0.1):
    total_rotate=0

    # Verify the axis:
    if axis != "X" and axis != "Y" and axis != "Z" :
        raise "MyException", "Can't use axis: " + axis

    while( total_rotate < 360.0 ) :
        # Issue a rotate command.
        if axis == "X":
            maestro.command("rotate", x=step)
        elif axis == "Y":
            maestro.command("rotate", y=step)
        elif axis == "Z" :
            maestro.command("rotate", z=step)
        # Redraw the window
        maestro.redraw()
        # Increment the total rotation by the step increment:
        total_rotate += step
        time.sleep(slp)

    return
```

When this is run as:

```
pythonrun spin.spin y 20 0.2
```

it spins the structure in the Workspace around the Y axis 360 degrees in increments of 20 degrees. The axis, increment, and the delay are all specified as parameters.

There are a couple of useful points to make about this script:

- In addition to including the `maestro` module we also include the standard Python module `time`. This allows us access to the `time.sleep()` function, used in this script to introduce a short delay after each rotation.
- Note how the `total_rotate` variable is used in the `while` loop. You can use any combination of parameters and variables in combination with the functions in the `maestro` module.
- In this particular function we need to use `maestro.redraw()` to force the contents of the Workspace to be redrawn after each rotation.

If you want to experiment with issuing commands from your own scripts, this example is a good place to start. As an exercise, create a script that rotates the structure first one way and then another.

It is also possible to use Maestro's command alias function to create new commands. For example, if you issue:

```
alias spin pythonrun spin
```

then you can just use `spin` like any other command.

In general, if a task can be performed by issuing a Maestro command, that is the preferred way to achieve it from a Python script. Not only does this generally result in the shortest possible script, but Maestro commands also automatically update the internal state of Maestro, redrawing the Workspace as needed, etc.

As powerful as these techniques are, there are still things you cannot achieve using Maestro commands alone. The following chapters describe how to control Maestro at a lower level of detail.

Manipulating the Workspace

Previous chapters covered how to run Python scripts in the Maestro environment and how to issue commands from those scripts. In this chapter we discuss more sophisticated and powerful features that allow you to directly manipulate the structure in the Workspace.

5.1 The Structure Concept

The chapters that follow refer to the concept of a *structure*. In Maestro, a structure is a collection of atoms. The contents of the Workspace are considered a structure, as are individual entries in the Project Table. Maestro stores information about structures: information about the atoms, bonds, and their properties. Python can get structures and manipulate them using this stored information. To do this, Python uses the functions in the `structure.py` module, which you can import into your script with the following command:

```
from schrodinger import structure
```

Once you get a structure, you can manipulate it in many ways, including deleting all the atoms.

5.2 Getting the Workspace Structure

As long as your script is running you can get the structure which corresponds to the contents of the Workspace. This structure is valid for as long as your script is running. However you should take care to get the structure again if your script issues a Maestro command that changes the contents of the Workspace, such as importing new structures or including new entries. The safest approach is to treat the structure as valid only until the completion of the next Maestro command.

To get the Workspace structure in a Python script, use a statement such as:

```
st = maestro.workspace_get()
```

Once you have a structure there are a number of operations you can perform on it.

5.3 Setting the Workspace Structure

Once you change the structure, you can pass it back to Maestro for display. To do this, use:

```
maestro.workspace_set()
```

Then you can use:

```
maestro.redraw_request()
```

to request that Maestro redraw the contents of the Workspace when your script has finished executing. You can also use `maestro.redraw()` to redraw the contents of the Workspace before your script has finished executing.

5.4 Operations on Structures

The following summarizes many of the operations possible with a structure object.

5.4.1 Obtaining Information on Atoms

You can obtain information on atoms through the atom property of a structure. It is possible to index individual atoms (indices start from 1) or to iterate over atoms. So the following two statements are equivalent:

```
for iatom in st.atom:
     print iatom.x

for iatom in range(1,len(st.atom)+1):
     print st.atom[iatom].x
```

Here `x` is the x-coordinate for the atom, but any of the properties shown in Table 5.1 and Table 5.2 can be used.

Table 5.1. Atom properties for access or modification.

Property	Description
`pdbname`	PDB atom name
`pdbres`	PDB residue name
`resnum`	PDB residue number and insertion code (returned as tuple)
`chain`	PDB chain name
`temperature_factor`	The PDB temperature factor
`atomic_number`	The atomic number
`x`	The x-coordinate
`y`	The y-coordinate
`z`	The z-coordinate

Table 5.1. Atom properties for access or modification.

Property	Description
`atom_type`	MacroModel atom type
`color`	Integer index representing color in Maestro color palette
`atom_name`	Maestro atom name (used for Jaguar mostly)
`partial_charge`	The partial atomic charge
`solvation_charge`	The solvation atomic charge
`formal_charge`	The formal charge
`secondary_structure`	The secondary structure assignment
`atom_style`	The molecular representation of the atom (wire, CPK etc.)
`visible`	The displayed/undisplayed flag

Table 5.2. Atom properties for access only - cannot be modified.

Property	Description
`entry_name`	The entry name
`molecule_number`	The number of the molecule to which this atom belongs
`number_by_molecule`	The atom number by molecule
`number_by_entry`	The atom number by entry

5.4.2 Obtaining Information on Bonds

Once you have an atom, you can obtain information on the bonds associated with that atom by using the bond property. So for example:

```
total_order=0
for iatom in st.atom:
     for ibond in iatom.bond:
          total_order += ibond.order
```

This example uses the bond order property. The available bond properties are listed in Table 5.3.

Table 5.3. Bond properties for access or modification.

Property	Description
`order`	Bond order
`atom1`	The first atom of the bond
`atom2`	The second atom of the bond
`style`	The molecular representation of the bond (wire, tube)

5.4.3 Adding and Deleting Bonds

This can be done via the structure class:

```
st.addBond( 12, 17,  2 )
```

adds (or sets if it already exists) a double bond between atoms 12 and 17.

```
st.deleteBond( 12, 17 )
```

deletes the bond between atom 12 and 17.

There is also the `areBound()` method which returns `True` if the two atoms specified have a bond between them.

5.4.4 Measuring and Adjusting

To measure, you can use the `measure()` method of the structure class:

```
st = maestro.workspace_get()
print st.measure( 1, 2 )    # Distance between atoms 1 and 2
print st.measure( 1, 2, 3 )    # Bond angle between 1, 2, and 3
print st.measure( 1, 2, 3, 4 ) # Torsion angle between 1, 2, 3, and 4
```

Adjust works in a similar manner:

```
st.adjust( 2.5, 1, 3 )    # Set distance between atoms 1 and 2 to 2.5 Angs
st.adjust( 110, 1, 2, 3 ) # Set bond angle between atoms 1, 2, and 3 to 110 degrees
st.adjust( 180.0, 1, 2, 3, 4 )
# Set the torsion angle between atoms 1, 2, 3, and 4 to 180.0 degrees.
```

5.4.5 Deleting Atoms

This is done with the `deleteAtoms()` method. For example:

```
to_del = [1,2,5]
st.deleteAtoms(to_del)
```

5.5 Things You Can Do with the Workspace Structure

There are many things you can do with a structure once you retrieve it. We can not cover all of them here. For more information, see the Structure Module reference document in:

`$SCHRODINGER/general/docs/py30_tutorial/schrodinger.structure.html`

What follows are a couple of examples.

Example 1. closecontact.py

This example illustrates how to issue Maestro commands and manipulate the Maestro Workspace structure. We use a Python function to highlight close contacts between any atoms separated by more than three bonds:

```
#closecontact.py
from schrodinger import maestro
from schrodinger import structure
from schrodinger import structureutil

def close_contacts(thresh=2.0) :
    # Get the current on-screen structure and calculate the number of atoms:
    st = maestro.workspace_get()
    distance = 0

    # Loop over all the atoms
    for iatom in st.atom :
        # Calculate a list of all the atoms which are not within three bonds
    # of the current "iatom". This is done by evaluating an ASL expression
    # "not( withinbonds 3 atom. iatom)"
        not_neighbours = structureutil.evaluate_asl( st,
                                                  "not( withinbonds 3 atom. %d)" %
                                                  int(iatom) )

        for jatom in not_neighbours :
            # Calculate the iatom-jatom distance
        distance = st.measure( iatom, jatom)
            if( distance < thresh ) :
        # This distance is less than the threshold - generate
        # a maestro command:
        maestro.command("distance %d %d"% (int(iatom), int(jatom)) )
```

By now you recognize the `import maestro` statement. This example also imports two additional modules, `structure` and `structureutil`, that we provide to perform operations at a lower level.

The first task in `close_contacts()` is to get the structure from the Workspace. Next we loop over all atoms in the structure.

For each atom in the Workspace, we only want to calculate the distance away from the current atom if it is more than three bonds away. An easy way to calculate this is with an ASL expression: `not( withinbonds 3 atom.`*iatom*`)`.

Evaluating this returns a Python list containing the atoms that satisfy our expression. We use that list in an inner loop to calculate the distance from the current atom. If a distance is less than the given threshold, we use Maestro to mark the distance by issuing the `distance` command.

This script contains common tasks: looping over all atoms, evaluating ASL expressions, and making measurements on the structure. Note that since we did not make any changes to the structure directly, we did not need to call `maestro.workspace_set()`.

Example 2. rotH.py

The following script adds hydrogens to the structure in the Workspace, and attempts to rotate the O-H and S-H groups on SER, THR, TYR, and CYS residues to place the hydrogen as close as possible to an acceptor. This is also an example of using multiple functions in a Python module. Only one of the functions is intended to be called from Maestro. The other simply improves the readability of the script.

Note: This script is provided purely as an example of how to manipulate the workspace—it is not intended to be a solution to the difficult problem of orienting hydrogens in proteins!

```
#rotH.py
#Import the modules we need:
from schrodinger import maestro
from schrodinger import structure
from schrodinger import structureutil

def rotH( radius = 4.0 ):
    """ This is the function which is to be called from Maestro. It takes
    a single parameter which is for each H-X how far from X  we should
    look for acceptors (the default is 4.0) """


    # Start by getting the workspace structure
    st = maestro.workspace_get()

    # Add hydrogens to the structure. We could do this by using a Maestro
    # command but this shows how we can also do it without using Maestro. After
    # we add hydrogens we can then get the number of atoms:
    structureutil.add_hydrogens( st )
    num_atoms = len(st.atom)
```

```
# Now locate all the rotatable atoms in all the CYS, SER, TYR and  THR.
# The simplest way to do this is using an ASL expression:
rotables = structureutil.evaluate_asl( st,
      "(res.ptype SER, THR, TYR, CYS ) and (atom.ptype HG, HG1, HH )")


# Create a couple of empty lists to hold the numbers of the hydrogen
# atoms which have nearby acceptors and the nearest acceptor for each one:
h_list = []
acc_list = []

# rotables is now a list of the appropriate atoms. We can loop over that
# and do what we need to do with them:
for h in rotables:
    # We need to locate a suitable dihedral angle to rotate:
    dihedral = get_dihedral_atoms( st, h )
    if not len(dihedral) == 4 :
        # Under normal circumstances there shouldn't be any situations
        # where we can't locate four atoms. However, there is the chance
        # that we might get an incomplete residue in a PDB file, so we'll
        # just ignore this residue if that's the case:
        continue



    # We can use another ASL expression to locate all the possible
    # acceptors within a reasonable distance of the heavy atom which
    # the hydrogen is attached to:
    acc_types = "OD1, OE1, OD1, SG, SD, ND1, NE2, OH, O"
    asl = "( within %f atom.num %d ) and atom.ptype %s" % ( radius,
                                                           dihedral[1],
                                                           acc_types)
    # Note that we exclude the backbone and the residue we are in:
    asl = \
    "( %s ) and ( sidechain or res. HOH ) and not fillres( atom.num %d )" \
    % ( asl,  dihedral[1] )
    acceptors = structureutil.evaluate_asl( st, asl )

    ang = 0.0
    min_dist = 1000.0
    best_ang = 0.0
    # If it actually found some suitable acceptors, we can
    # then scan the C-C-O-H dihedral see which dihedral gives the
    # closest contact to an acceptor. The scan is done in increments
    # of 5 degrees:
    while len(acceptors) > 0 and ang < 360.0 :
        # Set to the current angle:
        # Note that we pass the dihedral
        # as C2-C1-X-H as specifying it in this way indicates
```

```
            # we want to rotate the hydrogen:
            st.adjust( ang, dihedral[3], dihedral[2],
                       dihedral[1], dihedral[0] )


            # For each acceptor atom measure the distance and find out
            # whether we've actually found a closer match than we've found
            # before:
            for acc in acceptors:
                dist = st.measure( h, acc )
                if dist < min_dist:
                    min_dist = dist
                    best_ang = ang
                    best_acc = acc


            # increment the angle by 5 degrees:
            ang += 5.0


        # We've tried all the angles now. Reset it back to the one
        # which gave us the closest contact with an acceptor:
        if len(acceptors) > 0:
            st.adjust( best_ang, dihedral[3], dihedral[2],
                       dihedral[1], dihedral[0] )


            # Keep a list of the atoms associated with the best
            # interaction. For convenience we keep a string representation
            # of the atom number as we know now that we will later
            # transform this into an ASL expression:
            h_list.append( "%d" % h )
            acc_list.append( "%d" % best_acc )


    # Finally we tell Maestro that we want this structure to be used back
    # in the workspace:
    maestro.workspace_set( st )


    # Put up some hydrogen bond markers to highlight if we
    # have picked up any H-bonds
    if( len( h_list ) > 0 ):
        h_asl = ", ".join( h_list )
        acc_asl = ", ".join( acc_list )
        maestro.command( "hbondset1 atom.num %s , %s " % (h_asl, acc_asl) )


    return
```

```
#########################################################################
def get_dihedral_atoms( st, h ):
    """ For atom number h in the structure st, find four atoms to
    be used to scan the C-C-X-H dihedral. These are returned as a list.
    This function illustrates how to traverse the bonds of a structure """

    ret_list = []

    # The fist atom will be H itself:
    ret_list.append(h)

    # Now find the O or S attached to the H. Note that bonds (like everything
    # associated with structures) are indexed from "1" so we are
    # looking for the first bond:
    Xatom = st.atom[h].bond[1].atom2
    ret_list.append( int(Xatom) )

    # Now find a suitable non-H atom bonded to the X:
    C1atom = -1
    for b in st.atom[Xatom].bond:
        conn_atom = b.atom2
        # This is probably the easiest way to find a non-H atom:
        if not conn_atom.atomic_number == 1 :
            C1atom = int(conn_atom)
            ret_list.append( C1atom )
            # Leave the loop:
            break


    # Check to see that we did find a C1 atom:
    if C1atom < 0 :
        return ret_list

    # Now look for the final atom we need:
    for b in st.atom[C1atom].bond:
        conn_atom = b.atom2
        # This is probably the easiest way to find a non-H atom:
        if not conn_atom.atomic_number == 1 :
            C2atom = int(conn_atom)
            ret_list.append( C2atom )
            # Leave the loop:
            break



    return ret_list
```

Now that we have seen how to manipulate the structure from the Workspace, next we will look at how to access the Project Table directly.

Scripting the Project Table

While using Python scripts to manipulate the structure in the Workspace is useful for extending the functionality of Maestro, you can also automate Maestro by operating on structures in the Project Table. This chapter provides an overview of the possibilities.

6.1 Getting Information About the Project Table

Sometimes the easiest way to operate on the Project Table is to bring each structure into the Workspace in turn and operate on it there by issuing Maestro commands. You can do this with the `maestro.project_table_get()` function and looping over all the items in the table. By default such looping only returns the selected entries.

Example 1. saveimage.py

Here is an example that saves a `.jpg` image for each selected entry:

```
#saveimage.py

from schrodinger import maestro
from schrodinger import project


def save_image():

    pt = maestro.project_table_get()

    for sel_entry in pt:
        ename = pt[sel_entry]['s_m_entry_name']
        maestro.command("entrywsincludeonly entry \"%s\"" % ename )
        maestro.command("saveimage format=jpeg jpeg_quality=75 %s.jpg" %
                        ename )
```

Note: The entry name needs to be surrounded with quotes in the `entrywsincludeonly` command. This ensures that entry names that contain spaces are treated as one name.

If your project synchronization preferences are set to Automatic, each time you initiate an `entrywsincludeonly` on one entry in the Workspace, all entries in the project are updated. (To set these preferences, choose Preferences from the Maestro menu, then select the Project folder.) You can use this to make a change to all the selected entries.

Example 2. meth.py

The following script methylates amides in every selected structure by combining Maestro commands with direct manipulation of the Workspace structure for each selected entry:

```
#methylate.py

from schrodinger import maestro
from schrodinger import structureutil
from schrodinger import project


def methylate():

    pt = maestro.project_table_get()
    maestro.command("fragment organic Methyl")

    for sel_entry in pt:
        ename = pt[sel_entry]['s_m_entry_name']
        maestro.command("entrywsincludeonly entry",  ename )
        ct = maestro.main_ct_get()
        amides = structureutil.evaluate_smarts( ct, "[H]NC(=O)" )
        for amide in amides:
            maestro.command("attach %d" % amide[0] )
```

You can use practically any combination of Maestro commands and direct manipulations in the Workspace structure to achieve the results you want. We have also provided a similar function, `project.getPropertyNames()` that returns the property names.

6.2 Selecting Entries in the Project Table

Maestro has built-in support for selecting entries with `entryselectonly` and similar commands. This support is provided in three ways:

- Entry Selection dialog box
- ability to define filters
- use of ESL (see below)

There are limits to what type of selections can be generated with these features. They all rely on Maestro's Entry Selection Language (ESL). The ESL was designed to work on the properties associated with entries. For example:

```
entryselectonly property1 < 4
```

selects only the entries where `property1` has a value less than 4. However, since calculations are not possible in ESL, the following command would not work:

```
entryselectonly (2 * property1) < 4
```

While it is possible to create expressions of arbitrary complexity with the ESL, it is not possible to make selections in Maestro based on calculations performed on the actual structure (number of atoms etc.) nor to make selections based on functions of the entry properties such as the difference between two properties.

If your Python script needs to make selections in the Project Table that are not possible by issuing a `entryselectonly` command, the preferred method is a selection filter and the `project.selectRows()` function. To do this, write a simple Python function that is called for every entry in the project. If your function returns `True` the entry is selected, if it returns `False` the entry is not selected. Your function should accept two parameters: the CT corresponding to the current entry and a Python dictionary with an entry for each property in the Project Table. Note that it's also possible to use `project.selectRows()` with a list of row numbers to be selected. The following example demonstrates both these approaches:

Example 3. selring.py

This example selects entries that contain rings of a specified size:

```
#select_ring.py

from schrodinger import maestro
from schrodinger import structureutil
from schrodinger import project

def select_ring( ring_size ):

    pt = maestro.project_table_get()
    matches = []

    # Loop over all the entries in the project
    for row in xrange(1,len(pt)+1):
        rings = structureutil.find_rings(pt[row].structure)
        for ring in rings:
            if( len(ring) == ring_size ):
                matches.append(row)

    # Replacing them all through one call is quicker
    # than calling this method over and over
    pt.selectRows(project.REPLACE, rows=matches)

# The following two functions show how to select using a callback function
def select_ring2( ring_size ):

    pt = maestro.project_table_get()
    pt.selectRows(project.REPLACE, ring_size, function=myfunc)
```

```
def myfunc(project, ct, property_dict, *args):

    """
    Example callback function to select based on a property
    Return True if it should be selected
    Return False if it should be deselected
    """

   for ring in rings:
        if( len(ring) == ring_size ):
            return True
    return False
```

If you call this function from Maestro as:

```
pythonrun select_ring.select_ring 4
```

only those entries in the project with four-membered rings are selected.

Example 4. selprop.py

This example selects entries from the project based on a combination of the entry's structure and properties:. In this case we use the property dictionary to find the value of the property we are interested in.

```
#selprop.py

from schrodinger import maestro
from schrodinger import structureutil
from schrodinger import project

def select_proerty( ring_size ):

    pt = maestro.project_table_get()
    matches = []

    # Loop over all the entries in the project
    for row in xrange(1,len(pt)+1):
        num_atoms = pt[row].structure.atom_total
        if( num_atoms < 40 and  pt[row]['#stars'] != None and
            pt[row]['#stars'] > 4 ):
            matches.append(row)

    # Replacing them all through one call is quicker
    # than calling this method over and over.  So pass in the list
    # we built.
    pt.selectRows(project.REPLACE, rows=matches)
```

6.3 Working on Entries in the Project Table

As shown above, the `maestro.project_select_entry_get()` function can be used to bring each entry into the Workspace sequentially and operate on it there. While this is a useful technique, there may be times when this is not required, or where it would be inefficient to bring all entries into the Workspace. An alternative is to loop over all entries in the project table and operate on their structures directly.

Example 5. color_by_energy_gradient.py

Here is an example that sets the color of each entry, based on the relative molecular mechanics energy

Note: Because we return True from the function, the entry structure will be updated in the Project Table

```
#color_by_energy_gradient.py

from schrodinger import maestro
from schrodinger import structureutil
from schrodinger import project

def color_relative():

    """
    Use the property Realtive_Potential_Energy-OPLS-2005 to color all entries
    in  the project by the property.  Leaves entries without this property
    alone.
    """

    pt = maestro.project_table_get()

    for row in xrange(1, len(pt)+1):

        # Check to see we actually have the property for this
        # entry - if not, go to the next one.
        rel_energy = pt[row]['r_mmod_Relative_Potential_Energy-OPLS-2005']
        if rel_energy == None:
            print "Skipping entry %s since it has no value" % \
                  pt[row]['s_m_entry_name']
            continue
        else:
            # Color values are defined in $SCHRODINGER/maestro-vXX/data/colors.res
            col = 16 # red
            if rel_energy < 4.0:
                col = 4 # blue
            elif rel_energy < 8.0:
                col = 10 # green
```

```
        elif rel_energy < 12.0:
            col = 14 # orange

        num_atoms = pt[row].structure.atom_total
        ct = pt[row].structure
        for i in range(1, num_atoms+1):
            ct.atom[i].color = col
```

6.4 Adding New Columns to the Project Table

The previous example showed how you can loop over each entry in the project table and modify the structure. It is also possible to add new properties to the Project Table. To do this, simply add a new value to the property dictionary and make sure your function returns `True`.

Example 6. count_ch.py

The following example adds properties for the number of hydrogens and carbons to every entry in the project.

```
#count_ch.py

from schrodinger import maestro
from schrodinger import project

def count_ch( ):

    pt = maestro.project_table_get()
    matches = []

    for row in xrange(1, len(pt)+1):

        ct = pt[row].structure

        num_h_atoms = 0
        for a in ct.atom:
            if a.atomic_number == 1:
                num_h_atoms += 1

        num_c_atoms = 0
        for a in ct.atom:
            if a.atomic_number == 6:
                num_c_atoms += 1

        # Add or overwrite if already present
        # Format for name is: <type>_<author>_<property_name>
        # type can be i (integer), r (real), b (boolean), s (string)
        # author is "m" for maestro, "user" for user, etc.
        # property_name is any text. Underscores are allowed.
```

```
        pt[row]['i_user_Num_Carbons'] = num_c_atoms
        pt[row]['i_user_Num_Hydrogens'] = num_h_atoms

        pt.refreshTable()
```

Example 7. supersel.py

This is another example of looping over all entries. In this case an operation (superposition) is performed on the entries and the RMS deviation is added as a property in the Project Table.

```
#supersel.py
from schrodinger import maestro
from schrodinger import structureutil
from schrodinger import project

def superimpose_select():

    pt = maestro.project_table_get()
    matches = []

    # Ensure superposition can be done
    if pt.getSelectedRowTotal() < 2:
        return

    count = 1
    for sel_entry in pt:
        if count == 1:
            # First selected entry is the reference
            ref_entry = pt[sel_entry].structure
            num_atoms_ref = ref_entry.atom_total
            pt[sel_entry]['r_user_MyRMS'] = 0.0
        else:
            # For all others - superimpose non-hydrogen atoms
            ct = pt[sel_entry].structure
            num_atoms = pt[sel_entry].structure.atom_total
            if not num_atoms == num_atoms_ref :
                raise Exception, \
                      "There must be the same number of atoms in each structure"
            super_list = structureutil.evaluate_asl( ct, "not atom.ele H" )
            rms = structureutil.superimpose( ref_entry, super_list,
                                             ct, super_list )
            pt[sel_entry]['r_user_MyRMS'] = rms

        count = count + 1

    # Update the project table so users can see the added or
    # updated property column
    pt.refreshTable()
```

Running Jobs from Scripts

We have yet to provide an example of running a job from a Python script. It is in fact quite easy, either inside or outside Maestro. The script needs to simply issue the appropriate Maestro commands coupled with the `maestro.job_wait` function.

7.1 The maestro.job_wait Function

One limitation of Maestro command scripts (which are essentially just lists of Maestro commands) is that it is not possible to write a script that runs a job and waits for it to finish. This is easily overcome with a Python script and the `maestro.job_wait()` function. This example uses `glob.glob` to process all PDB files in the current working directory. However, it could have just as easily processed all the entries in a project as we saw earlier:

Example 1. allfiles.py

```
# allfiles.py
from schrodinger import maestro
from schrodinger import structureutil
import commands
import glob
import os


# This is the one that should be executed from Maestro. It uses the
# glob.glob subroutine to apply the action to all PDB files:
def mini_all_pdb() :
    # first clean up by removing any existing files:
    commands.getoutput( "rm *-min.pdb" )
    directory=os.getcwd()
    path = os.path.join(directory,"*.pdb")
    filelist = glob.glob( path )
    for afile in filelist:
        mini_pdb( afile )


# The function called for every PDB file:
def  mini_pdb(file) :
    import os
    #Convert the filename into a PDB code:
    ( path, fname ) = os.path.split( file )
    ( pdb_code, suffix ) = os.path.splitext( fname )
```

```
#Import the PDB file:
maestro.command("entryimport format=pdb %s" % file )
# Delete all molecules < 100 atoms and add hydrogens:
maestro.command("""
delete atom mol.atom < 100
hydrogenapply all
""" )

# Set up the MacroModel job, in vacuo, OPLSA2001, constrained CAs,
# and 100 iterations of LBFGS
maestro.command( """
potential  field=oplsaa
potential cutoff=normal
constrainedset atom.ptype " CA "
energytask mini
minienergy  method=lbfgs maxiter=100
jobsettings mmod incorporate=replaceentries jobname=%s
""" % pdb_code )

maestro.redraw()
# Run and wait for the job:
maestro.command("energystart")
maestro.job_wait(True)

maestro.command( "jobcleanup files=jobandmonitor %s "% pdb_code )

maestro.redraw()
#The job is now finished - export the structure to a new PDB file
out_file = pdb_code + "-min.pdb"
maestro.command("entryexport format=pdb source=selected %s" % out_file )

# Delete the entry from the project
maestro.command("entrydelete")
```

7.2 Running and Managing Jobs Outside Maestro

The current release of the Python modules adds some powerful tools for running and managing jobs outside Maestro. The `schrodinger.job.jobcontrol` module provides access to some of Schrödinger's job control functionality. It allows read access to the job database and the job host list, and can help with launching subjobs.

7.2.1 Access to the Job Database

Read-only access to the job database is provided by the `schrodinger.job.jobcontrol` class. A `Job` object can be created with a job ID string. These strings look like `isabel-0-434ac660` and are printed to the output of all Schrödinger jobs.

Once a `Job` object is created, the available keys can be listed with the `keys()` method and their values can be obtained as attributes of the object. The database values retrieved at creation time will never be updated automatically. They must be explicitly updated with the `readAgain` method.

This example from the interactive prompt demonstrates how to read the database, access attributes, and update your information:

```
>>> import schrodinger.job.jobcontrol as jobcontrol
>>> j = jobcontrol.Job("isabel-0-434ac660")
>>> j.keys()
['BackendFifo', 'BackendPid', 'ChildPid', 'Command', 'Dir', 'Envs',
'Home', 'Host', 'HostEntry', 'HostsFile', 'InputFiles', 'JobDB',
'JobDir', 'JobFifo', 'JobHost', 'JobId', 'JobPid', 'JobPort',
'JobUser', 'LaunchTime', 'LogFiles', 'MonitorInterval', 'Name',
'OutputFiles', 'Processors', 'Program', 'StartTime', 'Status',
'StatusTime', 'SubJobs', 'User']
>>> j.LaunchTime
'2005-10-10-15:52:00'
>>> j.LogFiles
['counterpoise.1146.blog']
>>> j.StatusTime
'2005-10-10-17:18:31'
>>> import time; time.sleep(1200) # wait a while
>>> j.StatusTime
'2005-10-10-17:18:31'
>>> j.readAgain()
```

7.2.2 Information on Job Hosts

A single function is provided to return a list of `Host` objects representing the information from the appropriate `schrodinger.hosts` file. The `Host` class provides attributes for processors, temporary storage (`tmpdir`), `SCHRODINGER` locations, and some less commonly needed pieces of information. (See the module documentation for more information.)

This example function lists all hosts with multiple processors:

```
def list_multiprocessor_hosts():
    import schrodinger.job.jobcontrol as jobcontrol
     for host in jobcontrol.get_hosts():
        if host.processors > 1:
           print "%20s: %d" % (host.name, host.processors)
```

7.2.3 Running Jobs From Python

The `jobcontrol.launch_job` function provides a way to run a Schrödinger job from a Python script. The single argument to this function is a Schrödinger command that you would issue from the command line (with the exception that `$SCHRODINGER` need not be included). This `launch_job` function returns a `Job` object.

This interactive example shows how to start a Jaguar job, do some other calculations, then wait for the Jaguar job to finish.

```
>>> import schrodinger.job.jobcontrol as jobcontrol
>>> job = jobcontrol.launch_job("jaguar run water.in")
>>> job.Status
'running'
>>> import time; time.sleep(10) # pretend this is useful
>>> job.wait()
>>> job.Status
'completed'
>>> job.OutputFiles
['water.01.in', 'water.out']
```

Writing Your Own Panels

All the scripts we have used so far have been run from the Maestro command line using the `pythonrun` command. It is, however, possible to write scripts that display their own graphical panels, similar to those of Maestro itself.

8.1 Tkinter

We support the Tkinter graphical user interface (GUI) toolkit for Python. It is simple to use and comes as a standard part of Python. This document describes how to use Tkinter with Maestro. Learning to program a GUI takes a bit of practice, but Tkinter and PMW make it relatively easy. For information on using Tkinter, see the recommended O'Reilly books on Python[3] or the following page on the Pythonware website[3] which offers a step-by-step tutorial. For information on PMW, see http://pmw.sourceforge.net[3]. You can also find more examples using Maestro and Tkinter at:

`$SCHRODINGER/python-v`*version*`/scripts/maestro/`

8.2 Important Considerations

When a Tkinter program is running within Maestro both programs need to share *event* information. Events in this context include mouse movements, mouse clicks, and key presses. Both Maestro and the Python/Tkinter script respond to events. In order for a Tkinter script running inside of Maestro to be able to share events with Maestro, special techniques are required.

First and foremost, never call `mainloop()` on any widget in your Tkinter script. If you do, your script will run, but Maestro will be inactive while your Tkinter widget is visible. This probably is not what you want since the real power of creating your own GUI panels is to allow them to interact with Maestro.

Instead of using `mainloop()` use `maestro.tk_toplevel_add()` to notify Maestro when you've built your panel and are ready to have it displayed. Maestro will then share the events it receives with your panel and the two can interact. When you are finished with your panel and want to dismiss it, call `maestro.tk_toplevel_remove()` to notify Maestro that you no longer need to share events.

3. Please see the notice regarding third party programs and third party Web sites on the copyright page at the front of this manual.

Note: Any number of Python/Tkinter scripts can be sharing events with Maestro.

Example 1. simple.py

Here is an example that displays a simple panel in Maestro:

```
# simple.py
from schrodinger import maestro
from Tkinter import *


simple_top = 0

def simple_quit_com( *ignore ):
    global simple_top
    maestro.tk_toplevel_remove( simple_top )
    simple_top.destroy()
    simple_top = 0


def simple():
    global simple_top
    # Don't put the panel up twice:
    if( simple_top != 0 ):
        return

    simple_top = Tk()
    quit_button = Button( simple_top, text='Quit', command = simple_quit_com)
    quit_button.pack()

    maestro.tk_toplevel_add(simple_top)
```

We use `maestro.tk_toplevel_add` to let Maestro know when we are ready to display the panel and `maestro.tk_toplevel_remove` when we are finished with it. Also notice the use of `simple_top` as a check to see if the panel is currently displayed. Before putting up a panel, it is generally good practice to see if the panel already exists. It can be confusing to have multiple instances of the same panel floating around.

8.3 Supporting Atom Selection from the Workspace

One of the most interesting things you can do with your panels is to receive information about which atoms are selected in the Workspace while your panel is active.

Note: If you request to receive selection information from Maestro, then no other panel within Maestro (or any other Python/Tkinter script) can receive selection information. By the same measure, if another panel starts receiving selection information (say it is opened from the main Maestro menu) then your panel loses the ability to receive selection information. This is a natural consequence of the way atom selection operates in the Maestro Workspace; picking information can only go to one place.

Example 2. simple_pick.py

Getting selections from the Maestro Workspace is simple. All you need to do is tell Maestro the name of the Python function you want called when a selection is received. Here we have extended the previous example by adding an additional function to receive atom selections. Remember to tell Maestro you no longer want to receive events when the panel is closed.

```
# simple_pick.py
from schrodinger import maestro
from Tkinter import *
from schrodinger import structure

simple_top = 0

def simple_quit_com( *ignore ):
    global simple_top
    maestro.tk_toplevel_remove( simple_top )
    maestro.picking_stop()
    simple_top.destroy()
    simple_top = 0

def simple_pick_cb( at ):
    st = maestro.workspace_get()
    pdb_res = st.atom[at].pdbres
    print "Picked residue is: %s " % pdb_res

def simple_pick():
    global simple_top
    # Don't put the panel up twice:
    if( simple_top != 0 ):
        return

    simple_top = Tk()
    quit_button = Button( simple_top, text='Quit', command = simple_quit_com)
    quit_button.pack()

    maestro.picking_atom_start( "Pick atom to have residue type printed",
                                "simple_pick.simple_pick_cb" )

    maestro.tk_toplevel_add(simple_top)
```

It is important to note that the function passed to `maestro.picking_atom_start()` must be fully qualified with the name of the function and the module, in this example, `simple_pick.simple_pick_cb()`.

The function that receives selections should be prepared to receive a single parameter, the atom number of the selected atom in the Workspace. Once you have the atom number you can use that to operate directly on the Workspace structure as shown in this example, or use it to issue Maestro commands.

8.4 Creating Panels with a Maestro Look and Feel

The Schrödinger Suite 2006 release of the Python tools introduces some additional modules—the `schrodinger.ui` package—that make it easier to create panels that better integrate with Maestro. In their simplest form, these modules provide interfaces to the standard Tkinter and PMW widgets, preconfigured to ensure that the appearance of the resulting panels closely matches that of Maestro itself. We strongly encourage you to use this package to build panels that interface to Maestro, because it avoids some of the problematic interactions between certain PMW widgets and Maestro.

Example 3. simple.py

Here's another version of Example 1, which uses the Schrödinger interface to Tkinter:

```
# simple.py
from schrodinger import maestro
import schrodinger.ui.widget as stk

simple_top = 0

def simple_quit_com( *ignore ):
    global simple_top
    maestro.tk_toplevel_remove( simple_top )
    simple_top.destroy()
    simple_top = 0


def simple():
    global simple_top
    # Don't put the panel up twice:
    if( simple_top != 0 ):
        return

    simple_top = stk.Tk()
    quit_button = stk.Button( simple_top, text='Quit',
    command = simple_quit_com)
    quit_button.pack()
```

```
maestro.tk_toplevel_add(simple_top)
```

There are a number of other facilities that are provided by these modules - we suggest you look at the reference documentation for more details.

Registering Python Functions with Maestro

Normally, the Python functions you write for use in Maestro will be called explicitly via the `pythonrun` command. However, there are some situations in which you may want to supply a Python function that will be called when particular events occur during the normal operation of Maestro. These functions, sometimes known as *callbacks*, are an important method for extending and modifying the default behavior of Maestro. You have already seen one example of a callback function at work in the previous chapter, where we used `maestro.picking_atom_start()` to register a Python function that was called by Maestro when an atom was selected in the Workspace. There are two additional callback functions that can be used to extend the capabilities of Maestro.

9.1 Periodic Functions

If you need to perform an action periodically, you can use:

`maestro.periodic_callback_add(`*your_callback_function_name*`)`

to register a Python function that will be called approximately 20 times a second. If you would like to perform the action at a lower frequency, maintain a counter in your function and only perform the action once every N times your callback function is called (i.e. every 20 to get about once a second).

Example 1. spin.py

Here is a simple example to show how this works:

```
#spin.py
# Register a periodic callback to spin the molecule
from schrodinger import maestro
def start_spin():
    maestro.periodic_callback_add( "spin.spin_cb");

# Unregister the callback:
def stop_spin():
    maestro.periodic_callback_remove( "spin.spin_cb" );

# The periodic callback:
def spin_cb():
   maestro.command("rotate y=5");
```

When you issue the `pythonrun spin.start_spin` command, the contents of the Workspace will be rotated around the Y-axis. This will continue until you explicitly issue the `pythonrun spin.stop_spin` command.

When you register a callback you should use the fully qualified `module.function` form. You can register as many periodic callback functions as you like during a Maestro session. When finished performing the periodic action, remember to unregister your callback function using:

`maestro.periodic_callback_remove(`*your_callback_function_name*`)`

Note: A registered callback function should not attempt to remove itself.

9.2 Mouse Hover Functions

You can also register a callback function, using:

`maestro.hover_callback_add(`*your_callback_function_name*`)`

that will be called by Maestro whenever the mouse is "hovering" (pointer is paused) over an atom in the Workspace. In this example we will demonstrate the mouse hover callback by utilizing the Maestro feature that allows atom-specific information to be displayed in the Workspace status bar.

Example 2. hover.py

The following example replaces the default string in the status bar with the atom number and partial charge of the atom the pointer is paused over:

```
#hover.py
from schrodinger import maestro
from schrodinger import structure
_last_atom = -1;

def set_hover():
    maestro.hover_callback_add( "hover.hover_cb" )

def clear_hover():
    maestro.hover_callback_remove( "hover.hover_cb" )

def hover_cb( at ):
    global _last_atom
    if( at == _last_atom ):
        return

    if at > 0 :
        st = maestro.workspace_get()
        pcharge = st.atom[at].partial_charge
```

```
        maestro.feedback_string_set("Atom: %d Charge = %5.3f" % (at, pcharge))

    _last_atom = at
    return;
```

Note: Register the callback function using the full `module.function` form. When you no longer need the mouse hover callback, unregister your function using:

`maestro.hover_callback_remove(`*your_callback_function_name*`)`

Debugging Your Scripts

Even if you are an experienced script writer, you are going to occasionally make mistakes. Now the task of debugging commences. This chapter discusses strategies for determining what has gone wrong with your script.

10.1 The Power of print

One of the simplest and at the same time most valuable Python debugging tools is the `print` statement. Since all the Python built-in types, including lists and dictionaries, can be printed, judicious placement of temporary `print` statements can often provide enough information to find even difficult errors. Note that the output resulting from a `print` statement appears in the window from which you started Maestro.

10.2 The pdb Module

Powerful as it is, sometimes the `print` statement is not going to give you enough information to find the problem. Fortunately, Python comes with a debugger (pdb) you can use even when your script is running in Maestro. You can find a complete description of pdb at www.python.org[4].

Example 1. spin_debug.py

It is easy to use pdb with your scripts. Simply create a special version of your main function that passes control to pdb, then call the modified function from Maestro. To illustrate, we are going to extend one of our previous examples:

```
#spin_debug.py
import pdb


from schrodinger import maestro
import time


def spin(axis="y",step=10,slp=0.1):
    total_rotate=0
```

4. Please see the notice regarding third party programs and third party Web sites on the copyright page at the front of this manual.

```
    if axis != "y" and axis != "x" and axis != "z" :
        raise Exception, "Can't use axis: " + axis

    while total_rotate < 360.0  :
        maestro.command("rotate %s=%d" % (axis,step))
        maestro.redraw()
        total_rotate += step
        time.sleep(slp)

    return

def spin_debug( *args ):
    pdb.run("spin.spin()")
```

Now, calling `spin_debug` from Maestro using `pythonrun` will run the script in the Python debugger, giving you access to all its debugging tools. From the `(Pdb)` prompt at the main window enter "`b spin.spin`" to set a breakpoint at the `spin.spin()` method. Next enter "`c`" to continue execution up to the breakpoint.

```
(Pdb) import spin

(Pdb) b spin.spin

Breakpoint 1 at /home/user/.schrodinger/maestroversion/scripts/
spin.py:23

(Pdb) c

> /home/user/.schrodinger/maestroversion/scripts/spin.py(23)spin()

-> total_rotate=0
```

Once you are at the breakpoint, use "`n`" to step line by line through the function and "`p`" to print current values of the variables. For a full list of available commands see the Python pdb website[5].

5. Please see the notice regarding third party programs and third party Web sites on the copyright page at the front of this manual.

The Maestro Scripts Menu

To this point, all our Python scripts have been run from the Maestro command line using `pythonrun`. While this works well during development and for occasional use, it can be quite cumbersome for frequently used, mature scripts. The Maestro Scripts menu offers a readily accessible home for all your frequently used scripts.

You can add scripts to this menu directly in Maestro (see Chapter 13 of the *Maestro User Manual* for details). Or you can add scripts to the Scripts menu manually by editing the `scripts.mnu` file as described below.

11.1 The scripts.mnu File

The key to adding your scripts to the Maestro Scripts menu is to add a new entry for each script to the `scripts.mnu` file. This file is located in the `.schrodinger/maestro`*version* directory in your home directory. You can create this file if it does not already exist.

The format for the `scripts.mnu` file is simple. There is one entry for each menu item. Each entry consists of two lines. The first describes the menu item itself. The second defines the command that will be run when that menu item is selected. In the next example we make our spin script available from the Scripts menu by creating a `$HOME/.schrodinger/maestro`*version*`/scripts.mnu` file containing the entry:

```
Spin
pythonrun spin.spin Y 30 0.1
```

The next time you start Maestro the Spin item will be available on the Scripts menu. Note how we have supplied arguments to the `spin.spin` command. This mechanism is not limited to running Python scripts. You can issue any Maestro command from the Scripts menu. Depending on your personal preferences, this can be an effective alternative to `pythonrun`.

11.2 Cascading Menus

Since there is a practical limit to the number of items you can place on a single menu, you probably will not want to add all your favorite scripts directly to the Scripts menu. By using cascading submenus you can not only organize related scripts, but in the process keep the top level Scripts menu more manageable. In this example we build a cascading submenu. We also illustrate how several entries in a single submenu can initiate the same command, each supplying different arguments. To define a cascading submenu, place a colon (`:`) in the first line of the entry:

```
Spin:X
pythonrun spin.spin X 10
Spin:Y
pythonrun spin.spin Y 10
Spin:Z
pythonrun spin.spin Z 10
```

Now we have a single Spin item in the top level Scripts menu. Spin contains a cascading submenu with items X, Y, and Z. Each item causes the contents of the Workspace to rotate about a different axis.

Note: You can only specify one level of cascading submenu items. You can not create an entry like the following:

```
Category:subcategory:item
```

11.3 Creating Scripts to be Installed in Maestro

As mentioned earlier it is possible to use Maestro to install scripts into the Scripts menu. To do this use the Manage... item in the Scripts menu. This works best when the script has been configured with some additional information which can be displayed in the installation dialog in Maestro. There are three things which are required to be added to the script to fully support being able to install it via Maestro:

1. A module-level doc string. This looks like:

   `__doc__= """`
   A description of the script
   Author
   Date
   `"""`

   This doc string will be displayed as the description of the script when the user selects the script in the Maestro script installation dialog box. It should be informative enough so that the user knows exactly what the script does.

2. A comment that begins with `#Name:` For example:

   ```
   #Name: Display all distances from an atom
   ```

   This is the text that will be displayed in the Scripts menu when the script is installed. The user has the option to edit this text at the point of installation but a useful default should be supplied.

3. A comment that begins with `#Command:`. For example:

```
#Command: pythonrun alldist.alldist
```

This is the command that will be used to run the script once is it installed in the Scripts menu.

Tips and Traps

12.1 Things to Watch Out For

This section is a collection of tips and techniques we have found useful. If you are having trouble making your script do what you want, you may find a solution here.

- The `maestro` module can only be used for scripts that are running inside Maestro. While the other modules like `structureutil` and `mm` can be used for scripts run with Python outside Maestro, the `maestro` module requires code that exists inside Maestro. (The next section describes a method for creating modules that can be used in a variety of situations.)
- Atoms and bonds are indexed from 1 (not 0) in functions that manipulate the structure.

12.2 Things That Might be Useful

Here are some things we have found useful:

- Emacs and Idle both provide many useful functions for creating and validating Python scripts. In Emacs, typing CTRL-C CTRL-C checks the syntax of the current file.
- A script that is executing within Maestro can be interrupted by typing CTRL+C in the terminal window in which Maestro was started.
- Modules are a great way to organize your code. When you create a set of functions you think are useful in multiple scripts, create a module to hold the scripts and place it in the module search path (for example `.schrodinger/maestro`*version*`/scripts` in your home directory). This makes the functions available for use in all your scripts.
- Even though the `maestro` module cannot be used outside Maestro, it is possible to write a module that includes `maestro`, and can be included as a module in another script or run as a stand-alone script. Here is an example:

```
# pdbname.py
def assign_pdb_names( ct, res_names = True, atom_names = True ):
     # Body not shown:

if __name__ == '__main__':
    #Running as stand-alone script:
    structureutil.for_all_structures( sys.argv[1], assign_pdb_names,
```

```
                        sys.argv[2], "Maestro", "Maestro", True, True)
else:
    # Are we running inside Maestro?
    try:
        from schrodinger import maestro
        def pdbname():
            st = maestro.workspace_get()
            assign_pdb_names(st)
            maestro.workspace_set(st)
    except
        pass
```

This script can be used as follows:

- From within Maestro: `pythonrun pdbname.pdbname`
- As a stand-alone script:

  `$SCHRODINGER/utilities/python pdbname.py` *infile outfile*
- Imported into another module: `import pdbname`

Running Scripts Outside Maestro

13.1 Running Your Scripts

Although it is not possible to use any of the methods from the `maestro` module when running outside Maestro, it is possible to use the `structure` and `structureutil` modules.

Note: It is best to use the version of Python we supply (`$SCHRODINGER/utilities/python`) as this ensures that all the necessary environment variables are set up before your script is run.

The main difference between running a script inside or outside Maestro is how you gain access to structures. When running a script inside Maestro, you can get structures from the Workspace or the project. However, in your stand-alone scripts you need to read the structures directly from the files. Fortunately this is easy. Short filter scripts can be created that read structures from one file and write modified versions to another file.

13.2 Simple Filters

Reading structures is best done with the `StructureReader` class. The structure reader knows how to read from PDB, SD (mdl), or Maestro files. If the format is not specified explicitly, the file suffix is used to determine the format:

`.ent` `.pdb`	PDB
`.sd` `.sdf`	SD
`.mae`	Maestro

Once you have a structure object, it knows how to write or append itself to a specified file. Again, the format can be PDB, SD, or Maestro. If the format is not specified explicitly, the file suffix is used.

Example 1. Read a file and write a new file

Here is an example of a filter that reads a file and writes to a new file all the structures that contain a chlorine atom:

```
from schrodinger import structure
import sys
import os

os.remove(sys.argv[2])

for s in structure.StructureReader(sys.argv[1], format='maestro'):
    for iatom in s.atom:
        if iatom.atomic_number == 17:
            s.append( sys.argv[2], format="maestro")
            break
```

To run this script, enter the following command:

`$SCHRODINGER/utilities/python findcl.py` *infile*`.mae` *outfile*`.mae`

Note: In this case, the format is specified explicitly as `format=maestro` so this only operates on Maestro files, regardless of the file suffix. Any number of operations could be performed on the structure before it is written out again.

There is also a mechanism for examining the properties of the structures in the file. For example, you could examine those quantities that are calculated and added to a file by programs like Glide, QikProp, or MacroModel. The "property" field of the structure class can be used as a Python dictionary to get and set properties associated with that structure.

Note: When operating from files, you must use the property name exactly as it appears in the file. Also if you create a property name, it must conform to the following prefix convention, based on the type of data you wish to add:

Real number	`r_user_`
Integer	`i_user_`
String	`s_user_`
Boolean	`b_user_`

The following example creates a new integer property. When the output file is imported into Maestro, this new property appears in the Project Table as "My Sum".

```
s.property['i_user_My_Sum']= sum
```

Example 2. Properties from a Glide poseviewer file.

The following example shows how to use the properties from a Glide poseviewer file.

Note: No explicit format is given, so the suffix of the input and output file determine the format. The receptor (first structure) is omitted and the remaining ligand poses are only written to the input file if the GlideScore is < -4.5.

```
from schrodinger import structure
import sys
import os

os.remove(sys.argv[2])

cnt = 1
for s in structure.StructureReader( sys.argv[1] ):
    if cnt != 1:
        if s.property['r_i_glide_gscore'] < -4.5 :
            s.append(sys.argv[2])
    cnt += 1
```

Getting Help

Schrödinger software is distributed with documentation in PDF format. If the documentation is not installed in `$SCHRODINGER/docs` on a computer that you have access to, you should install it or ask your system administrator to install it.

For help installing and setting up licenses for Schrödinger software and installing documentation, see the *Installation Guide*. For information on running jobs, see the *Job Control Guide*.

Maestro has automatic, context-sensitive help (Auto-Help and Balloon Help, or tooltips), and an online help system. To get help, follow the steps below.

- Check the Auto-Help text box, which is located at the foot of the main window. If help is available for the task you are performing, it is automatically displayed there. Auto-Help contains a single line of information. For more detailed information, use the online help.
- If you want information about a GUI element, such as a button or option, there may be Balloon Help for the item. Pause the cursor over the element. If the Balloon Help does not appear, check that Show Balloon Help is selected in the Help menu of the main window. If there is Balloon Help for the element, it appears within a few seconds.
- For information about a panel or the folder that is displayed in a panel, click the Help button in the panel. The Help panel is opened and a relevant help topic is displayed.
- For other information in the online help, open the Help panel and locate the topic by searching or by category. You can open the Help panel by choosing Help from the Help menu on the main menu bar or by pressing CTRL+H.

  To view a list of all available Maestro–related help topics, choose Maestro from the Categories menu of the Categories tab. Double-click a topic title to view the topic.

If you do not find the information you need in the Maestro help system, check the following sources:

- *Maestro User Manual*, for detailed information on using Maestro
- *Maestro Command Reference Manual,* for information on Maestro commands
- Frequently Asked Questions pages, at https://www.schrodinger.com/Maestro_FAQ.html

The manuals are also available in PDF format from the Schrödinger Support Center. Information on additions and corrections to the manuals is available from this web page.

If you have questions that are not answered from any of the above sources, contact Schrödinger using the information below.

E-mail:	help@schrodinger.com
USPS:	101 SW Main Street, Suite 1300, Portland, OR 97204
Phone:	(503) 299-1150
Fax:	(503) 299-4532
WWW:	http://www.schrodinger.com
FTP:	ftp://ftp.schrodinger.com

Generally, e-mail correspondence is best because you can send machine output, if necessary. When sending e-mail messages, please include the following information, most of which can be obtained by entering `$SCHRODINGER/machid` at a command prompt:

- All relevant user input and machine output
- Python purchaser (company, research institution, or individual)
- Primary Python user
- Computer platform type
- Operating system with version number
- Python version number
- Maestro version number
- mmshare version number

Reference Modules

The Schrödinger installation of Python includes a `schrodinger` package, that contains a number of other packages and modules. The descriptions of these packages and modules are in HTML format and are located in the following directory:

`$SCHRODINGER/docs/general/python/py30_tutorial`

Links to these documents are provided in the HTML version of this manual. You can also open the file `schrodinger.html` in this directory and navigate to the documentation from there. The main packages are described in Table A.1.

Table A.1. Description of supplied modules.

Module	Description
`application`	Application-specific functions
`infra`	Low-level package with tools not intended for general script usage. We reserve the right to change these modules as we feel necessary.
`job`	Functions for launching and managing jobs
`maestro`	Functions used for interaction with Maestro
`project`	Functions used with the Project Table, either inside or outside Maestro
`structure`	Functions to read, write, and manipulate structures
`structureutil`	Functions that operate on structure objects: finding rings and matching SMARTS expressions, etc.
`ui`	Graphical user interface tools. Fixes some bugs in Tkinter and PMW that cause bad interactions with Maestro. Use these to ensure consistent look and feel with Maestro. Lastly, provides reusable specialized components.

Copyright Notices

NCSA HDF5 Software Library and Utilities

6. THIS SOFTWARE IS PROVIDED BY THE UNIVERSITY AND THE CONTRIBUTORS “AS IS” WITH NO WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED. In no event shall the University or the Contributors be liable for any damages suffered by the users arising out of the use of this software, even if advised of the possibility of such damage.

Portions of HDF5 were developed with support from the University of California, Lawrence Livermore National Laboratory (UC LLNL). The following statement applies to those portions of the product and must be retained in any redistribution of source code, binaries, documentation, and/or accompanying materials:

This work was partially produced at the University of California, Lawrence Livermore National Laboratory (UC LLNL) under contract no. W-7405-ENG-48 (Contract 48) between the U.S. Department of Energy (DOE) and The Regents of the University of California (University) for the operation of UC LLNL.

DISCLAIMER: This work was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately-owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

C and C++ Libraries for Parsing PDB Records

The C and C++ libraries for parsing PDB records are Copyright (C) 1989 The Regents of the University of California. All rights reserved.

Redistribution and use in source and binary forms are permitted provided that the above copyright notice and this paragraph are duplicated in all such forms and that any documentation, advertising materials, and other materials related to such distribution and use acknowledge that the software was developed by the University of California, San Francisco. The name of the University may not be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED “AS IS” AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.

120 West 45th Street
32nd Floor
New York, NY 10036

101 SW Main Street
Suite 1300
Portland, OR 97204

3655 Nobel Drive
Suite 430
San Diego, CA 92122

Dynamostraβe 13
68165 Mannheim
Germany

SCHRÖDINGER